HW02-Q4

Lab02-Q9

Q4: Church numerals

The logician Alonzo Church invented a system of representing non-negative integers entirely using functions. The purpose was to show that functions are sufficient to describe all of number theory: if we have functions, we do not need to assume that numbers exist, but instead we can invent them.

Your goal in this problem is to rediscover this representation known as Church numerals. Here are the definitions of zero, as well as a function that returns one more than its argument:

def zero(f):

return lambda x: x

def successor(n):

return lambda f: lambda x: f(n(f)(x))

First, define functions one and two such that they have the same behavior as successor(zero) and successsor(successor(zero)) respectively, but do not call successor in your implementation.

Next, implement a function church\_to\_int that converts a church numeral argument to a regular Python integer.

Finally, implement functions add\_church, mul\_church, and pow\_church that perform addition, multiplication, and exponentiation on church numerals.

def one(f):

"""Church numeral 1: same as successor(zero)"""

"\*\*\* YOUR CODE HERE \*\*\*"

def two(f):

"""Church numeral 2: same as successor(successor(zero))"""

"\*\*\* YOUR CODE HERE \*\*\*"

three = successor(two)

def church\_to\_int(n):

"""Convert the Church numeral n to a Python integer.

>>> church\_to\_int(zero)

0

>>> church\_to\_int(one)

1

>>> church\_to\_int(two)

2

>>> church\_to\_int(three)

3

"""

"\*\*\* YOUR CODE HERE \*\*\*"

def add\_church(m, n):

"""Return the Church numeral for m + n, for Church numerals m and n.

>>> church\_to\_int(add\_church(two, three))

5

"""

"\*\*\* YOUR CODE HERE \*\*\*"

def mul\_church(m, n):

"""Return the Church numeral for m \* n, for Church numerals m and n.

>>> four = successor(three)

>>> church\_to\_int(mul\_church(two, three))

6

>>> church\_to\_int(mul\_church(three, four))

12

"""

"\*\*\* YOUR CODE HERE \*\*\*"

def pow\_church(m, n):

"""Return the Church numeral m \*\* n, for Church numerals m and n.

>>> church\_to\_int(pow\_church(two, three))

8

>>> church\_to\_int(pow\_church(three, two))

9

"""

"\*\*\* YOUR CODE HERE \*\*\*"

Use Ok to test your code:

python3 ok -q church\_to\_int

python3 ok -q add\_church

python3 ok -q mul\_church

python3 ok -q pow\_church

Q9: I Heard You Liked Functions...

Define a function cycle that takes in three functions f1, f2, f3, as arguments. cycle will return another function that should take in an integer argument n and return another function. That final function should take in an argument x and cycle through applying f1, f2, and f3 to x, depending on what n was. Here's what the final function should do to x for a few values of n:

n = 0, return x

n = 1, apply f1 to x, or return f1(x)

n = 2, apply f1 to x and then f2 to the result of that, or return f2(f1(x))

n = 3, apply f1 to x, f2 to the result of applying f1, and then f3 to the result of applying f2, or f3(f2(f1(x)))

n = 4, start the cycle again applying f1, then f2, then f3, then f1 again, or f1(f3(f2(f1(x))))

And so forth.

Hint: most of the work goes inside the most nested function.

def cycle(f1, f2, f3):

"""Returns a function that is itself a higher-order function.

>>> def add1(x):

... return x + 1

>>> def times2(x):

... return x \* 2

>>> def add3(x):

... return x + 3

>>> my\_cycle = cycle(add1, times2, add3)

>>> identity = my\_cycle(0)

>>> identity(5)

5

>>> add\_one\_then\_double = my\_cycle(2)

>>> add\_one\_then\_double(1)

4

>>> do\_all\_functions = my\_cycle(3)

>>> do\_all\_functions(2)

9

>>> do\_more\_than\_a\_cycle = my\_cycle(4)

>>> do\_more\_than\_a\_cycle(2)

10

>>> do\_two\_cycles = my\_cycle(6)

>>> do\_two\_cycles(1)

19

"""

"\*\*\* YOUR CODE HERE \*\*\*"

Use Ok to test your code:

python3 ok -q cycle

3.3 Write a function that takes in a list and returns the maximum product that

can be formed using nonconsecutive elements of the list. The input list will

contain only numbers greater than or equal to 1.

def max\_product(s):

"""Return the maximum product that can be formed using non-consecutive

elements of s.

>>> max\_product([10,3,1,9,2]) # 10 \* 9

90

>>> max\_product([5,10,5,10,5]) # 5 \* 5 \* 5

125

>>> max\_product([])

1

"""

if s == []:

return 1

elif len(s) == 1: # Base case optional

return s[0]

else:

return max(max\_product(s[1:]), s[0] \* max\_product(s[2:]))

(b) Define the following function so that it properly identifies mountain numbers. A mountain number is a

number that either

i. has digits that strictly decrease from right to left OR strictly increase from right to left

ii. has digits that increase from right to left up to some point in the middle of the number (not necessarily

the exact middle digit). After reaching the maximum digit, the digits to the left of the maximum

digit should strictly decrease.

def check\_mountain\_number(n):

"""

>>> check\_mountain\_number(103)

False

>>> check\_mountain\_number(153)

True

>>> check\_mountain\_number(123456)

True

>>> check\_mountain\_number(2345986)

True

"""

def helper(x, is\_increasing):

if x // 10 == 0:

return True

2

if is\_increasing and (x % 10) < ((x // 10) % 10):

return helper(x // 10, is\_increasing)

return (x % 10) > ((x // 10) % 10) and helper(x // 10, False)

return helper(n, True)

2.8 Consider the subset sum problem: you are given a list of integers and a number k.

Is there a subset of the list that adds up to k? For example:

>>> subset\_sum([2, 4, 7, 3], 5) # 2 + 3 = 5

True

>>> subset\_sum([1, 9, 5, 7, 3], 2)

False

>>> subset\_sum([1, 1, 5, -1], 3)

False

Note: You can use the in operator to determine if an element belongs to a list:

>>> 3 in [1, 2, 3]

True

>>> 4 in [1, 2, 3]

False

def subset\_sum(seq, k):

if len(seq) == 0:

return False

elif k in seq:

return True

else:

return subset\_sum(seq[1:], k - seq[0]) or \

subset\_sum(seq[1:], k)

In the above diagrams, every node has a “strong” child and a “weak” child, and

primary stress is placed on the leaf that has the greatest number of strong parents.

In the spirit of computational linguistics, let’s write a function that, given one of

these tree structures, identifies the stressed part of a word or phrase. 1

def primary\_stress(t):

"""

>>> word = tree("", [

tree("w", [tree("s", [tree("min")]), tree("w", [tree("ne")])]),

tree("s", [tree("s", [tree("so")]), tree("w", [tree("ta")])])])

>>> primary\_stress(word)

'so'

>>> phrase = tree("", [

tree("s", [tree("s", [tree("law")]), tree("w", [tree("degree")])]),

tree("w", [tree("requirement")])])

>>> primary\_stress(phrase)

'law'

"""

def helper(t, num\_s):

if is\_leaf(t):

return [label(t), num\_s]

if label(t) == "s":

num\_s = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

return max([\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_],

key = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

return \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def helper(t, num\_s):

if is\_leaf(t):

return [label(t), num\_s]

if label(t) == "s":

num\_s = num\_s + 1

return max([helper(b, num\_s) for b in branches(t)], key = lambda a: a[1])

return helper(t, 0)[0]

2.2 Write a function that takes in no arguments and returns two functions, prepend and

get, which represent the “add to front of list” and “get the ith item” operations,

respectively. Do not use any python built-in data structures like lists or dictionaries.

You do not necessarily need to use all the lines.

def nonlocalist():

"""

>>> prepend, get = nonlocalist()

>>> prepend(2)

>>> prepend(3)

>>> prepend(4)

>>> get(0)

4

>>> get(1)

3

>>> get(2)

2

>>> prepend(8)

>>> get(2)

3

"""

get = lambda x: "Index out of range!"

def prepend(value):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

f = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def get(i):

if i == 0:

return value

return \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

return \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

get = lambda x: "Index out of range!"

def prepend(value):

nonlocal get

f = get

def get(i):

if i == 0:

return value

return f(i - 1)

return prepend, lambda x: get(x)